Goethe Universität Frankfurt am Main

Fachbereich 12 Mathematik und Informatik
Institut für Informatik

Bachelorarbeit

Eine Untersuchung zu Monte-Carlo Tree Search mit Anwendung auf MAXSAT

Kerem Bozyel

Eingereicht bei: Prof. Dr. Manfred Schmidt-Schauss

Professur für Künstliche Intelligenz und Softwaretechnologie

Wintersemester 2016/2017

Eingereicht am: 08.05.2017

Erklärung

Erklärung gemäß Bachelor-Ordnung Informatik 2011 §25 Abs. 11

Hiermit erkläre ich, dass ich die Bachelorarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe.

Frankfurt am Main, den 08.Mai.2017

_____

(Kerem Bozyel)

Inhaltsverzeichnis

Zusammenfassung

Seit dem Spiel Go gewinnt der Monte-Carlo Tree Search (MCTS) im Bereich der künstlichen Intelligenz immer mehr an Bedeutung. Wichtig dabei ist, dass der MCTS approximativ optimale Züge durch Monte-Carlo Simulationen findet. Je höher die Anzahl der Simulationen, desto genauer wird die Approximation für einen optimalen Zug. In dieser Arbeit wird der MCTS mit Anwendung auf das Maximum Satisfiability Problem, kurz MAXSAT, untersucht. Das MAXSAT-Problem handelt von der maximalen Anzahl an erfüllbaren Klauseln in einer Klauselmenge. Bei sehr großen Klauselmengen ist es sehr aufwendig, die Lösung exakt zu bestimmen. Dazu wird in dieser Arbeit ein neuer Ansatz, namens UCTMAXSAT[6], vorgestellt, welches das MAXSAT-Problem mithilfe von MCTS löst.

Kapitel 1

Einführung

1.1 Künstliche Intelligenz und Spiele: Mensch gegen Computer

Bis heute gibt es zahlreiche Wettkämpfe zwischen Computer und Mensch. Einer der ersten großen Wettkämpfe war der Wettkampf im Jahr 1996 zwischen dem vom IBM entwickelten Schachcomputer „Deep Blue“ und Garri Kasparow. Garri Kasparow war Weltmeister in Schach von $1985 - 2000$[19]. Seit dieser Zeit sind Computer in der Lage mithilfe von Heuristiken und Baumsuchen Spiele zu gewinnen. Zu den klassischen Spielbaumsuchen zählen beispielsweise der Minimax-Algorithmus und die Alpha-Beta-Suche. Diese Suchverfahren werden in Spielen wie Schach, Vier Gewinnt etc[15] genutzt. Das Problem bei diesen Suchverfahren tritt meistens auf, wenn das Spielfeld zu groß ist, wie zum Beispiel bei dem Brettspiel Go. Das 19x19 große Spielfeld erschwert die klassischen Verfahren, denn im ersten Zug gibt es 361 Auswahlmöglichkeiten[15]. Aus diesem Grund wurde für das Spiel Go eine andere Baumsuche von einem Computerprogramm Alpha Go verwendet, welches einen optimalen Zug im Spielbaum mithilfe von Simulationen approximiert. Der Name dieses Suchverfahrens lautet Monte-Carlo Tree Search (kurz: MCTS). Wie sich im Jahre 2016 herausstellte, gewann Alpha Go gegen Lee Sedol mit 4 zu 1 in 5 Partien[7]. Mit diesem Sieg zeigt sich das MCTS als ein erfolgreiches Suchverfahren für Spiele wie Go. In den späteren Kapiteln dieser Arbeit wird auf die Funktionsweise dieses Suchverfahrens genauer eingegangen.

1.2 Überblick und Aufbau

Diese Bachelorarbeit beschäftigt sich mit der Untersuchung von MCTS mit Anwendung auf MAXSAT. Sie ist in 5 Kapitel aufgeteilt, wobei das erste Kapitel als Einleitung dient. Im zweiten Kapitel werden Grundlagen bezüglich zu MCTS erläutert. Anschließend wird im dritten Kapitel auf den Monte-Carlo Tree Search eingegangen. Hierbei wird der Algorithmus und die Funktionsweise des MCTS dargestellt. Das vierte Kapitel gibt einen Überblick zu dem MAXSAT-Problem.
Das letzte Kapitel widmet sich zum Hauptschwerpunkt beziehungsweise Ziel dieser Arbeit. Dabei werden bisherige MAXSAT Algorithmen vorgestellt und diese werden mit dem neuen Ansatz UCTMAXSAT aus Goffinet und Ramanujan verglichen. Außerdem wird der Algorithmus, die Baumstruktur und dessen Anwendung auf MAXSAT im Detail beschrieben und unter bestimmten Aspekten wie Laufzeit und Speicheraufwand untersucht. Als Fazit sind die Rückschlüsse der Anwendung von MCTS auf MAXSAT aufgeführt.

Kapitel 2

Grundlagen

2.1 Spieltheorie und Entscheidungstheorie

2.1.1 Spieltheorie

Die Spieltheorie analysiert Spiele, in denen die Spieler versuchen, den optimalen Zug zu wählen, wodurch der eigene Gewinn maximiert wird. Dennoch ist es nicht immer der Fall, dass der gewünschte Gewinn erreicht wird, das heißt nicht alle Strategien der Spieler garantieren ihnen effizienten Gewinn[17]. Darum muss sich der Spieler Gedanken machen, welchen Zug er spielen soll, sodass für ihn eine ideale Gewinnstrategie zustande kommt.

2.1.2 Normalform und Extensivform

Im Folgenden wird auf nicht-kooperative Zwei-Personen-Spiele beschränkt und auf den Fakt, dass die Spieler rationale Entscheidungen treffen sollen. Rational bedeutet, dass jeder Spieler immer den bestmöglichen Zug spielen soll, damit der Gewinn maximal ist. Es gibt zwei Formen von Spielen:

1. Spiel in Normalform
2. Spiel in Extensivform

Bei einem Spiel in Normalform werden die Strategien in einer Matrix festgehalten. Dadurch kann man die beste Antwort unabhängig davon, welchen Zug der Gegenspieler trifft, herausfinden. Im Gegensatz zur Normalform wird im Spiel in Extensivform das Spiel sequentiell gespielt, das heißt, dass die Spieler nacheinander spielen und somit andere Strategien wählen müssen, um ihren Gewinn zu maximieren. Jedoch gibt es bei Spielen in Extensivform verschiedene Informationsverteilungen.

Man unterscheidet zwischen vollständigen und unvollständigen Informationen[17]. In Spielen mit vollständigen Informationen ist es jedem Spieler bewusst, welche Entscheidungen in einem bestimmten Spielzustand ein beliebiger Spieler treffen kann und welche Auszahlung ansteht. Beispiele für solche Spiele sind Schach, Dame, Vier Gewinnt und vieles mehr. Hingegen sind Spiele wie Poker und Skat Spiele mit unvollständigen Informationen. In der Künstlichen Intelligenz wird zwischen vollständig beobachtbar und teilweise beobachtbar unterschieden, welches die Umgebung eines Spiels beschreibt[15].

2.1.3 Entscheidungstheorie

Wenn man Spiele wie Schach und Poker vergleicht, so erkennt man, dass die Umgebungen verschiedene Eigenschaften haben. Betrachtet werden vorerst nur die Eigenschaften *„deterministisch vs. stochastisch“*[15]. Stochastisch bedeutet, dass eine Aktion zufällig auftritt. Dadurch gibt es beim Entscheiden für eine Aktion eine gewisse Unsicherheit. Mit diesem Problem beschäftigen sich die Markov-Entscheidungsprozesse (kurz: MDP, Englisch: *Markov-Decision Process*).
Ein MDP besteht aus einem 5-Tupel (S, A, T, R, γ) und ist laut Russell et al.[14] wie folgt definiert:

- S ist die endliche Menge an Zuständen
- A ist die endliche Menge an Aktionen
- $T(s'|s, a)$ beschreibt die Übergangswahrscheinlichkeit eines Zustand s in einen Zustand s' durch eine Aktion a
- $R(s, a)$ gibt die Belohnung für den Übergang, gegeben einer Aktion a, in den aktuellen Zustand s an
- γ ist ein Diskontfaktor, welches die Belohnungen $R(s, a)$ abzinst[1]

Ziel ist es, durch eine Reihe von Aktionen den größten Gewinn zu erhalten. Diese Reihe von Aktionen wird bezeichnet als π. Markov-Entscheidungsprozesse bilden eine Grundlage für Monte-Carlo Tree Search, denn MDP findet man in Spielen mit Extensivform und das Entscheiden unter Unsicherheit spielt bei MCTS eine große Rolle.

[1] In der Spieltheorie werden künftige Belohnungen bestraft. Mehr dazu in Quelle [11] (Kapitel Spieltheorie)

2.2 Monte-Carlo Methoden

Bei Suchverfahren wie Minimax ist es bei Spielen mit großen Spielbäumen sehr schwer berechenbar, da beim Minimax-Algorithmus der ganze Spielbaum traversiert werden muss. Beispielsweise beim Spiel Tic Tac Toe gibt es $9! = 362.880$ mögliche Zugfolgen, wobei man von einem leeren Spielfeld aus das Spiel startet[15]. Aus diesem Grund werden für Spiele mit großen Spielbäumen eine andere Bewertungsfunktion benötigt. In den folgenden zwei Unterabschnitten werden zwei Bewertungsfunktionen vorgestellt.

2.2.1 Monte-Carlo Evaluation

Bei der Monte-Carlo Evaluation (auf Deutsch: *Monte-Carlo Bewertung*) werden die Knoten anhand von Playouts bewertet. Ein Playout ist eine Folge aus zufälligen Zügen[13]. In anderen Worten heißt es, dass bei einer bestimmten Spielkonfiguration beziehungsweise einem Knoten im Spielbaum bis zu einem Blatt zufällige Züge gewählt werden, um damit dann einen Gewinner oder einen Verlierer zu ermitteln. Die Knoten werden anhand der Anzahl der Playouts bewertet. Beispielsweise bewertet man jeden Knoten i nach der Gewinnerquote, das heißt, wie viel Prozent der gesamten Anzahl an Playouts gewonnen wurden. Ein Nachteil dieser Bewertung ist es, dass es sehr aufwendig ist, denn der Weg eines Playout im Baum ist linear zur Tiefe des Baumes[13].

2.2.2 K-armiger Bandit

Man stelle sich vor, dass man vor mehreren Spielautomaten sitzt und gerne spielen möchte. Nach einigen Spielen stellt man fest, dass man unter diesen n-Automaten einen oder eine bestimmte Anzahl l an Automaten bevorzugt, da diese Automaten den höchsten Gewinn eingebracht haben. Es kann dennoch sein, dass beim Weiterspielen die nicht-bevorzugten Automaten einen höheren Gewinn einbringen könnten als die bisher-bevorzugten Automaten. Dieses Problem bezeichnet man auch als Exploitation-Exploration Dilemma[10]. Wichtig hierbei ist es, dass nicht nur bevorzugte Automaten betrachtet werden sollten, sondern auch jene Automaten, die nicht bevorzugt werden. Damit beschäftigt sich das Multi-Armed Bandit Problem (kurz: *MAB* oder *K-armiger Bandit*)[20]. Ein Spieler i mit K-Armen versucht denjenigen Arm $l \in K$ zu wählen, sodass Spieler i den optimalen Gewinn erhält.

2.2.3 Upper Confidence Bounds

Anhand des *Upper Confidence Bounds* (kurz: UCB) ist es möglich, das Problem des K-armigen Banditen zu lösen. Die Formel des UCB lautet wie folgt[1]:

$$UCB(j) = \bar{X}_j + \sqrt{\frac{2 \cdot ln(n)}{n_j}}$$

Das $\bar{X}_j$ bezeichnet den Gewinn, der im Durchschnitt durch Betätigen eines Arms oder Spielen eines Automaten j erreicht wird. Die Variable n_j beschreibt die Anzahl der Simulationen einer Maschine und n stellt die Anzahl aller Simulationen von allen Maschinen dar.
Für die Herleitung von $\sqrt{\frac{2 \cdot ln(n)}{n_j}}$ wird die *Chernoff-Hoeffding Ungleichung*[21] benötigt, welche jedoch hier nicht behandelt wird, da diese Herleitung komplex ist.

Kapitel 3

Monte-Carlo Tree Search

3.1 Motivation

Das Spiel Go besteht aus einem $19x19$ großen Feld. Würde man beispielsweise den Minimax-Algorithmus auf dieses Spiel ausführen, so ist diese Berechnung auf so einem großen Feld unmöglich . 2016 war für Alpha Go ein großer Erfolg. Die Nutzung des Monte-Carlo Tree Search brachte Alpha Go den Sieg gegen Lee Seedol ein[7].
Die Besonderheit an Monte-Carlo Tree Search ist, dass MCTS die Monte-Carlo Evaluation verwendet, in dem ein Suchbaum aufgebaut wird und anhand von Monte-Carlo Simulationen der Suchbaum erweitert wird[13]. Wie oben beschrieben gibt es sogenannte Playouts, welche eine Folge von zufälligen Zügen ist. Durch diese Playouts werden Züge bewertet und es wird ein Zug approximiert, der für den Spieler optimal sein soll. Dennoch muss man mit einer gewissen Fehlerwahrscheinlichkeit rechnen, da diese Playouts durch zufällige Wahlen von Zügen bestehen. Dies unterscheidet sich von klassischen Suchverfahren, bei denen es eher statische Bewertungsfunktionen gibt[13].
Im Folgenden wird das Prinzip des MCTS vorgestellt. Daraufhin wird auf den Baum und die Baumtraversierung, welche aus vier Schritten besteht, eingegangen. Im Anschluss werden Heuristiken zu Monte-Carlo Tree Search vorgestellt.

3.2 Baumtraversierung

Den MCTS gibt es in vielen Varianten, wobei diese Varianten dennoch einen grundlegenden Aufbau haben. Dieser Aufbau besteht aus zwei Strategien[8]:

1. Verwalten des Suchbaums (*internal tree*)
2. Durchführen eines Playouts (*search policy*)

Im ersten Fall geht es um das interne Speichern des Suchbaums. Ausgehend von einem Startknoten wird ein Kindknoten ausgewählt und dieser zum Suchbaum hinzugefügt. Anschließend tritt die zweite Strategie *search policy* ein. Bei dieser Strategie wird an diesem ausgewählte Knoten ein Playout gespielt. Es werden von diesem Knoten aus bis zu einem Blatt oder zu einer Endspielsituation zufällige Züge gewählt. Anschließend werden alle Werte der Knoten im Suchbaum neu aktualisiert. Ebenfalls gibt es einen Grenzwert an Simulationen, sodass ab einer bestimmten Anzahl an Simulationen die beste Aktion ausgegeben wird und diese dann ausgeführt wird. Diese beiden Strategien bestehen somit aus vier grundlegenden Schritten, welches im Weiteren genauer erläutert werden.

Selektion

Mithilfe einer Evaluationsfunktion (Heuristik), folgt die Auswahl eines Knotens s. Dennoch ist zu beachten, wie schon beim K-armigen Bandit beschrieben, dass nicht-bevorzugte Knoten auch betrachtet werden sollen (Stichwort: Exploration-Exploitation-Dilemma)[8]. Denn in manchen Fällen kann es sein, dass die nicht-bevorzugten Knoten eventuell einen höheren Wert haben.

Expansion

Bei diesem Schritt betrachtet man die Kinder des ausgewählten Knotens s. Falls ein oder mehrere Kinder existieren, welche noch nicht expandiert wurden, so werden diese Kinder dem Suchbaum hinzugefügt. Falls jedoch alle Kinder des Knotens s expandiert wurden, so wird ein anderer oder auch ein Kindknoten s' selektiert und der Expansionsschritt wird dann auf dem Knoten s' ausgeführt[4].

Simulation

Ab diesem Schritt folgt die *search policy*. Hierbei werden auf den neu hinzugefügten Kindknoten Playouts ausgeführt. Ebenfalls gibt es für das Playout zwei Möglichkeiten:

- Man wählt rein-zufällige Züge ohne die Spielregeln zu kennen
- Man nutzt eine Heuristik, sodass weniger Zufall vorhanden ist (Beispiel: Zwickmühle beim Schach (In [4] bezeichnet als *„proximity to the last move"*). Die letzten Züge betrachten, wodurch man vorher weiß, ob man verloren/gewonnen hat).

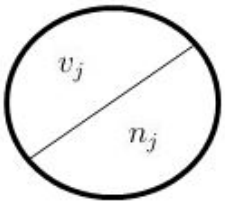
Kurz gefasst heißt es, dass vom neu hinzugefügten Kindknoten bis zu einem Blatt zufällige Züge gewählt werden.

Backpropagation

Zum Schluss tritt der letzte Schritt Backpropagation ein. Nachdem durch die Simulation ein Blatt erreicht wurde, werden alle Knoten im Suchbaum aktualisiert. Außerdem wird bei jedem durchlaufenen Knoten j die Simulationszahl n_j um eins erhöht[8].

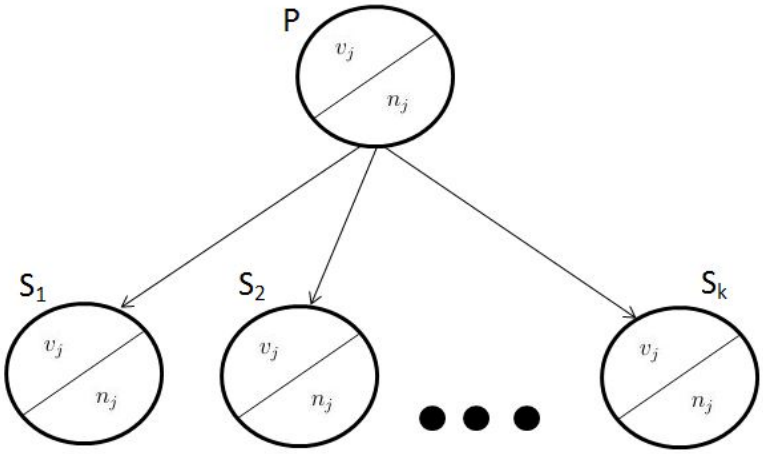
Beispiel 1

Ein Knoten beinhaltet zwei Variablen: v_j und n_j, wobei v_j den durchschnittlichen Gewinn und n_j die Anzahl an Simulationen von Knoten j ausdrückt. Ein Knoten j sieht folgendermaßen aus[1]



[1]Die folgenden Abbildungen in diesem Beispiel wurden mit Microsoft Word erstellt

Gegeben sei folgender Baum:



1. Selektion

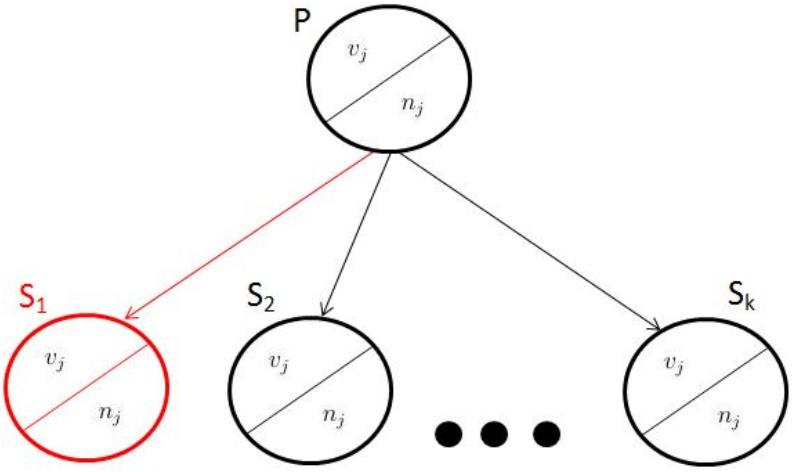


Abbildung 3.1: Wahl eines Knotens mithilfe einer Heuristik

2. Expansion

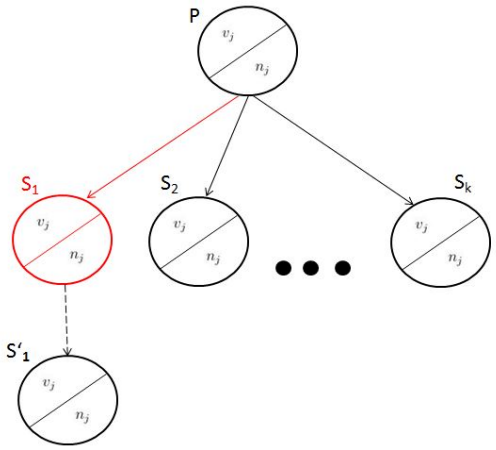


Abbildung 3.2: Kindknoten s'_1 wird dem Suchbaum hinzugefügt

3. Simulation

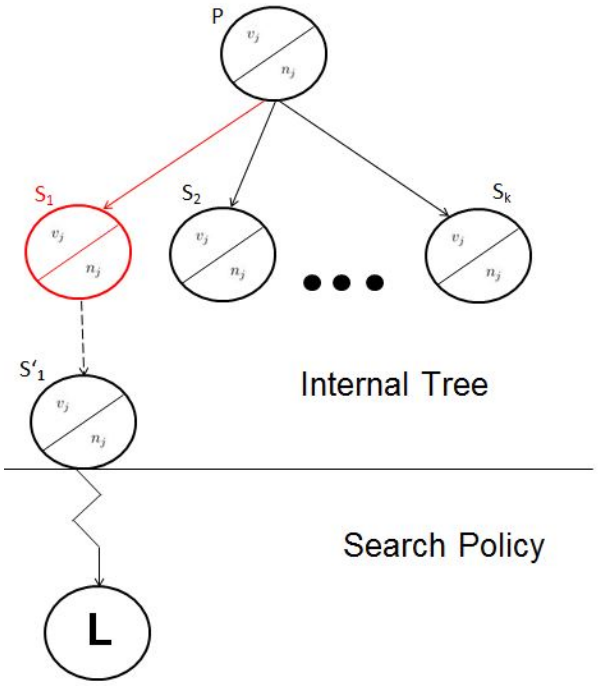


Abbildung 3.3: $L \in \{-1, 0, 1\}$ ist das Ergebnis eines simulierten Spiels

4. Backpropagation

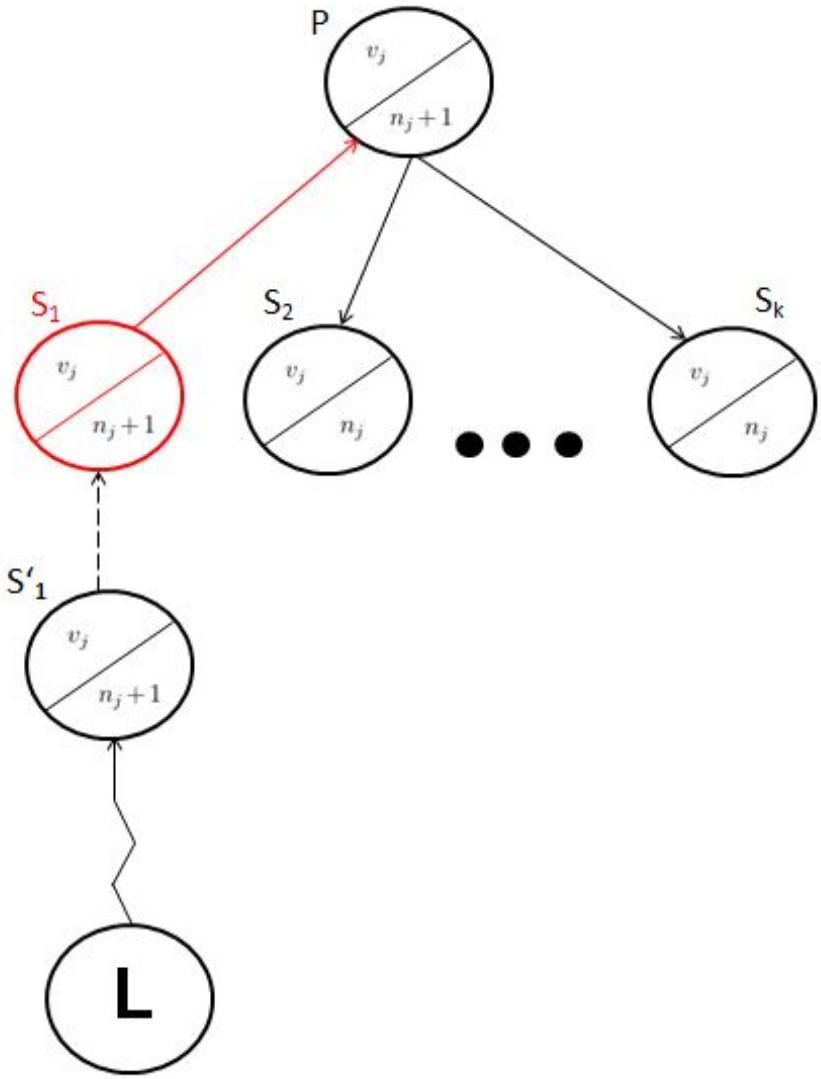


Abbildung 3.4: Aktualisieren der Werte. Ebenfalls wird auch v_j neu berechnet

3.3 Heuristiken

In diesem Abschnitt werden drei Heuristiken vorgestellt, wobei UCT für die spätere Untersuchung eine große Rolle spielt. Für die anderen beiden Heuristiken folgt ein kurzer Überblick.

3.3.1 UCT

Der *Upper Confidence Bounds for Tree* (kurz: UCT) beruht auf dem Upper Confidence Bounds in Kombination mit MCTS. Wichtig hierbei ist, dass durch einen Extraparameter C das Gleichgewicht zwischen bevorzugten (Exploitation) und nichtbevorzugten (Exploration) Armen j gesteuert werden kann. Im $MCTS$ kommt der UCT beim Selektionsschritt zum Einsatz. Denn es wird der Kindknoten j mit dem größten UCT-Wert ausgewählt. Die Formel für UCT sieht folgendermaßen aus[4]:
Sei J die Menge aller Kindknoten eines Knoten p.

$$u(j) = argmax_{j \in J} \left(v_j + C \cdot \sqrt{\frac{ln(n_p)}{n_j}} \right)$$

Hierbei steht für n_p die Anzahl an Simulationen des Knotens p und n_j ist die Anzahl der Simulationen eines Kindknotens $j \in J$. Außerdem beschreibt v_j den durchschnittlichen Gewinn eines Kindknotens j.

Beispiel 2

Annahme: $C = 2$

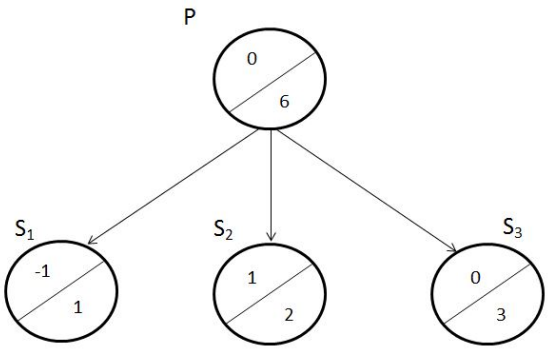


Abbildung 3.5: Wahl eines Knotens mithilfe der UCT-Heuristik

$\Rightarrow$ Knoten S_2 wird ausgewählt, da $S_2 = 2,893 > S_1 = 1,677 > S_3 = 1,55$

3.3.2 AMAF

Mithilfe der *All Moves As First* Heuristik (kurz: $AMAF$) kann man den UCT-Algorithmus optimieren. Hierbei ist es wichtig, dass jede Aktion a ausgehend von einem Knoten p mit gleicher Wichtigkeit betrachtet wird, das heißt alle anderen Aktionen sind genauso gleichgültig wie die erste Aktion. Der AMAF-Wert ist der Durchschnitt der simulierten Spiele ausgehend von Knoten p. Formell sieht es wie folgt aus[4]:

- $S_i(p,a) \in \{-1,0,1\}$ Ergebnis eines i-ten simulierten Spiels ausgehend von Knoten p mit Aktion a, wobei -1 für Verloren, 0 für Unentschieden und 1 für Gewonnen steht
- $AMAF_p = \frac{1}{n} \cdot \sum_{i=1}^{n} S_i(p,a)$, wobei n die Anzahl an simulierten Spielen beschreibt.

Mit dieser Heuristik wird für jeden Knoten p zusätzlich der $AMAF$-Wert gespeichert, der eine Gewinn-Rate darstellt.

3.3.3 RAVE

Der *Rapid Action Value Estimate* (kurz: $RAVE$) ist eine Zusammensetzung des UCT und der AMAF Heuristik. Mittels eines Koeffizienten β kann die Balance zwischen UCT und AMAF gesteuert werden[4]. Nach der Modifikation der Formel aus 3.3.1 (UCT) sieht sie nun so aus:

$$k \in argmax_{j \in J} \left((1 - \beta(n_p)) \cdot \left(v_j + C \cdot \sqrt{\frac{ln(n_p)}{n_j}} \right) + \beta(n_p) \cdot AMAF_p \right)$$

$$\text{mit } \beta(n_p) = \sqrt{\frac{b}{3 \cdot n_p + b}}$$

Mit dem Koeffizienten b wird die Anzahl an benötigten Traversierungen gesteuert, sodass die $RAVE$ Schätzung verbessert wird. Je höher der Koeffizient β, desto mehr Einfluss nimmt der AMAF-Wert bei der Wahl eines Knotens beim Selektionschritt.

Kapitel 4

MAXSAT

4.1 Erfüllbarkeitsproblem

Um einen booleschen Ausdruck auf Erfüllbarkeit zu testen, ist es die einfachste Variante, dies anhand einer Wahrheitstafel zu lösen[16]. Eine Wahrheitstafel enthält alle möglichen Belegungen der Variablen einer Formel φ, um damit dann φ auszuwerten.

A	B	A $\wedge$ B	A $\vee$ B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Sei n die Anzahl an Variablen. In der obigen Tabelle hat n den Wert 2 und Formeln dieser Größe können in annehmbarer Zeit ausgewertet werden. Nun ist das Problem, dass wenn n wächst, die Laufzeit exponentiell zunimmt, denn für jede Variable gibt es zwei Werte 0 und 1. Das heißt bei n-Variablen ergeben sich $2 \cdot 2 \cdot 2 \cdot ... = 2^n$ viele mögliche Belegungen. Dieses Problem ist auch unter dem Namen SAT bekannt und ist wie folgt definiert:

$$SAT = \{\varphi | \text{ Aussagenlogische Formel } \varphi \text{ ist erfüllbar}\}$$

Das Entscheidungsproblem SAT findet man in verschiedenen Formen wie KNF-SAT und k-SAT[1] .

$$KNF\text{-}SAT = \{\varphi | \text{ Aussagenlogische Formel } \varphi \text{ ist erfüllbar und liegt in konjunktiver Normalform } (CNF) \text{ vor}\}$$

$$k\text{-}SAT = \{\varphi | \text{ Aussagenlogische Formel } \varphi \text{ liegt in } CNF \text{ vor und besitzt höchstens } k \text{ Literale}\}$$

[1] Mehr zu KNF-SAT und k-SAT in Quelle[18]

KNF-SAT ist NP-vollständig. Ein Problem ist NP-vollständig, wenn es mithilfe einer *nichtdeterministischen Methode* in polynomieller Zeit lösbar ist[18]. Hingegen sind alle $k \geq 3$ bei k-SAT wie zum Beispiel 3-SAT NP-vollständig.
Das *Maximum Satisfiability* Problem (kurz: $MAXSAT$) ist eine Variante von k-SAT, worauf aber in Abschnitt 4.3 genauer eingegangen wird.

4.2 Definitionen

In diesem Abschnitt werden wesentliche Elemente der Aussagenlogik vorgestellt. Die Inhalte stammen aus der Quelle [16].

4.2.1 Logische Notationen

Die Aussagenlogik dient zur Modellierung von Wissen aus der realen Welt. Es werden Gegebenheiten und Zusammenhänge aus dem Alltag formell beschrieben. Eine aussagenlogische Formel φ besteht aus Variablen, die einen booleschen Wert *wahr* oder $falsch$ annehmen können, und den klassischen logischen Operatoren wie $\neg, \wedge, \vee, \Rightarrow$ und $\Leftrightarrow$. Boolesche Werte sind 0 und 1, wobei 0 für $falsch$ und 1 für *wahr* steht. 0 und 1 werden auch als Konstanten bezeichnet. Ein Atom ist entweder eine Konstante oder eine Variable X. Hingegen ist ein Literal ein Atom, welches entweder negiert oder nicht-negiert vorliegt. Literale werden für Normalformen benötigt, welches im nächsten Unterabschnitt genauer erläutert wird.

Beispiel 3

Gegeben ist folgende Aussage:

Peter kommt zur Party genau dann, wenn Olaf und Anne auch dabei sind.

Dazu werden die Variablen definiert:

$$P := \text{Peter}$$
$$O := \text{Olaf}$$
$$A := \text{Anne}$$

Daraus ergibt sich dann folgende aussagenlogische Formel:

$$P \Leftrightarrow O \wedge A$$

4.2.2 Normalformen

Eine aussagenlogische Formel φ kann auf verschiedene Arten und Weisen dargestellt werden. Dies kann in zwei Formen vorliegen:

- Disjunktive Normalform (DNF)
  1. Bilde Konjunktionen von Literalen L_i
  2. Danach bilde Disjunktionen aus den Konjunktionen von Schritt 1
- Konjunktive Normalform (CNF)
  1. Bilde Disjunktionen von Literalen L_i
  2. Anschließend bilde Konjunktionen aus den Disjunktionen von Schritt 1

In dem untenstehenden Beispiel wird CNF und DNF in ihrer Anwendung dargestellt.

Beispiel 4

1. CNF:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_1 \vee x_3)$$

2. DNF:

$$(\neg x_1 \wedge \neg x_2) \vee (x_1 \wedge x_2)$$

Die konjunktive Normalform wird auch als $Klauselnormalform$ bezeichnet und wird meistens in Mengenform betrachtet[16]. Eine Disjunktion von Literalen wird auch als Klausel bezeichnet. Damit ist eine CNF eine Menge von Klauseln. Für das obrige Beispiel sehen die Klauseln dann wie folgt aus:

$$\{\{x_1, x_2, \neg x_3\}, \{\neg x_2, x_1, x_3\}\}$$

Da die Operatoren $\vee$ und $\wedge$ assoziativ und kommutativ sind, ist es möglich, die konjunktive Normalform als Menge zu betrachten. Wie schon in Abschnitt 4.1 beschrieben, kann es zu einer gewissen Komplexität kommen, um eine aussagenlogische Formel φ in CNF-Form auf Erfüllbarkeit zu testen. Denn bei einer CNF müssen alle Klauseln den Wert 1 haben, sodass φ erfüllbar ist.

4.3 Maximum Satisfiability Problem

Das *Maximum Satisfiability Problem* (kurz: MAXSAT) beschäftigt sich mit der maximalen Anzahl an erfüllbaren Klauseln[2]. Es ist nicht unbedingt wichtig, ob eine Klauselmenge C erfüllbar oder unerfüllbar ist.
Sei F eine Klauselmenge mit $\{C_1, ..., C_n\}$ und X die Menge der vorhandenen Variablen aus F, wobei C_j mit $1 \leq j \leq n$ eine Klausel darstellt. Die Belegungen der Variablen aus der Menge X werden festgehalten mit der Funktion $\rho : X \to \{0, 1\}$. Um zu wissen, dass bei einer bestimmten Belegung ρ die Klausel C_j erfüllbar oder nicht erfüllbar ist, wird folgende Funktion π definiert:

$$\pi(\rho, C_j) = \begin{cases} 1, falls & C_j = 1 \\ 0, sonst. & \end{cases}$$

Das heißt die Funktion gibt genau dann eine 1, wenn die Klausel C_j durch die Belegungen der Variablen ρ erfüllt wird. Die Anzahl der erfüllbaren Klauseln wird bezeichnet als S und sie wird als Summe zusammengefasst:

$$S = \sum_{j=1}^{n} \pi(\rho, C_j)$$

Damit das MAXSAT-Problem gelöst werden kann, wird die maximale Anzahl an erfüllbaren Klauseln benötigt:

$$S^* = \max \sum_{j=1}^{n} \pi(\rho, C_j)$$

Der Unterschied zwischen SAT und MAXSAT ist es, dass bei MAXSAT viel mehr Belegungen zulässig sind. Denn bei SAT spielen Belegungen keine Rolle, die die jeweilige Formel nicht erfüllen.

Beispiel 5

Gegeben sei folgende CNF φ:

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee x_2)$$

oder in Klauselnormalform

$$\{\{x_1, \neg x_2\}, \{\neg x_1, \neg x_2\}, \{\neg x_1, x_2\}, \{x_1, x_2\}\}$$

Für jegliche Belegungen ergibt sich folgende Tabelle:

x_1	x_2	φ	Anzahl erfüllter Klauseln
0	0	0	3
0	1	0	3
1	0	0	3
1	1	0	3

$\Rightarrow$ Die maximale Anzahl an erfüllbaren Klauseln beträgt 3 und kann durch alle Belegungen erreicht werden. Ebenfalls erkennt man, dass die Formel zwar unerfüllbar ist, aber jedoch 3 Klauseln erfüllt sind.

Kapitel 5

Anwendung auf MAXSAT

Ziel dieses Abschnittes ist die Untersuchung des Monte-Carlo Tree Search Suchverfahrens und die Bewertung dessen Anwendung auf MAXSAT. Dabei wird auf einige Aspekte wie Speicher, Laufzeit usw. eingegangen.
Zunächst werden bisherige MAXSAT Solver beschrieben. Anschließend wird ein neuer Ansatz UCTMAXSAT aus [6] vorgestellt. Zum Schluss steht die Analyse zu MCTS.

5.1 MAXSAT-Solver

Es gibt einige SAT-Solver, die für das MAXSAT Problem angepasst wurden und Erfolg erwiesen, worunter WalkSAT und Novelty[6] zu erwähnen sind. MAXSAT-Solver sind in zwei Kategorien aufgeteilt: vollständige und unvollständige Solver. Der Unterschied zwischen beiden Solverkategorien ist es, dass unvollständige Solver meistens eine zufallsbasierte Suche verwenden, um optimale Lösungen zu finden. Hingegen verwenden vollständige Solver Methoden wie Branch and Bound[6]. Unvollständige Solver beginnen damit, dass jede Variable einer gegebenen Klauselmenge $F = \{C_1, ..., C_n\}$ mit einem zufälligem Wert aus $\{0, 1\}$ belegt wird und iterativ Variablen geflippt[1] werden. Diese werden auch *Stochastic Local Search* (kurz: SLS) Algorithmen genannt[6].
Da sich der Monte-Carlo Tree Search mit Playout und Entscheidungen unter Unsicherheit (siehe Kapitel 2 Abschnitt 2.1.3) beschäftigt, liegt das Augenmerk in diesem Kapitel auf den unvollständigen Solvern.

[1] Eine Variable x „flippen“ bedeutet, dass der ursprüngliche Wert invertiert wird, das heißt wenn x den Wert 1 hat, dann hat x nach dem Flip den Wert 0

5.1.1 SLS Algorithmen

In praktischer Anwendung setzen sich die SLS Algorithmen erfolgreich durch, um das SAT-Problem zu lösen. Spezifische SLS Algorithmen, wie WalkSAT und Novelty[6], wurden für das MAXSAT Problem verwendet. Ein SLS Algorithmus beginnt damit, dass für eine gegebene Klauselmenge $F = \{C_1, ..., C_n\}$ eine zufällige Belegung φ gewählt wird. Danach wird eine Variable geflippt, wodurch anschließend wieder getestet wird, ob F erfüllbar ist. Dies entspricht in etwa folgendem Verfahren:

Input: Klauselmenge F, maximale Anzahl an Versuchen k, maximale Anzahl an Flips m
Output: Belegung φ, wodurch F erfüllt wird

1. Solange $k > 0$
   (a) Wähle eine zufällige Belegung φ der Variablen (also jede Variable mit einem zufälligen Wert aus $\{0, 1\}$ belegen)
   (b) Solange $m > 0$
      i. Falls F erfüllt ist, dann stoppe und gebe φ zurück
      ii. Ansonsten wähle mithilfe einer Heuristik eine Variable v aus
      iii. $\varphi = \varphi$ mit geflippter Variable v
      iv. $m = m - 1$
   (c) $k = k - 1$
2. Falls F nicht erfüllt, dann existiert keine Lösung

Um dies auf das MAXSAT Problem anzuwenden, benötigt es einige Modifikationen. Es wird zusätzlich für jede Variable v_i aus F mit $1 \leq i \leq n$ die Anzahl an erfüllten Klauseln gespeichert t_i und zum Schluss wird der größte Wert t_i zurückgegeben.
Ein großer Nachteil kann sein, dass in manchen Fällen im vorherigen Schritt $m - 1$ eine geflippte Variable v_i im Schritt m erneut geflippt wird. Dies wird als *Zyklenproblem* bezeichnet. Dies muss vermieden werden, da sonst der Algorithmus hängen bleibt.

5.1.2 WalkSAT

Ein einfacher SLS Algorithmus ist WalkSAT[5]. Beim WalkSAT Algorithmus wird zufällig eine unerfüllte Klausel C_j aus der Klauselmenge $F = \{C_1, ..., C_n\}$ ausgewählt. Im vorherigen Abschnitt wurde stattdessen zufällig eine Variable v_i aus F ausgewählt. Anschließend wird bei WalkSAT eine Variable $v_{i,j} \in C_j$ zufällig gewählt. Dadurch wird die Wahrscheinlichkeit geringer, dass eine Variable $v_{i,j}$ erneut geflippt wird.
Um es verständlicher zu machen, sieht die Modifikation von WalkSAT[9] für das MAXSAT Problem folgendermaßen aus:

Input: Klauselmenge F, maximale Anzahl an Versuchen k, maximale Anzahl an Flips m, Liste L (enthält Paare der Form (b, Anzahl der erfüllten Klauseln))
Output: Belegung b, wobei die Anzahl an erfüllten Klauseln maximal ist

1. Solange $k > 0$
   (a) Wähle eine zufällige Belegung b der Variablen (also jede Variable mit einem zufälligen Wert aus $\{0, 1\}$ belegen)
   (b) Solange $m > 0$
      i. Falls F erfüllt ist, dann gib $|F|$ zurück
      ii. Ansonsten
         A. c = zufällige Wahl einer unerfüllten Klausel
         B. v = zufällige Wahl einer Variablen aus c
         C. $b' = b$ mit geflippter Variable v
         D. Speichere in $L \rightarrow$ (b', Anzahl der erfüllten Klauseln)
         E. $m = m - 1$
   (c) $k = k - 1$
2. Gib die Belegung b zurück, welches in L die höchste Anzahl an erfüllten Klauseln hat.

Beim WalkSAT Algorithmus wird das Zyklenproblem zum Teil minimiert, da jetzt die Auswahl einer Variable $v_{i,j}$ geringer ist, weil zuerst eine unerfüllte Klausel C_j gewählt wird. Dennoch kann es zu dem Fall kommen, dass das Zyklenproblem auftritt.

Beispiel 6

Gegeben sei folgende Klauselmenge F:

$$\{\{\neg v_1, v_2\}, \{v_1, v_2\}\}$$

$\Rightarrow$ Es kann sein, dass der Zufall immer die Variable v_1 trifft und somit bildet sich ein Zyklus.

5.1.3 Novelty

Ähnlich wie der WalkSAT Algorithmus beruht Novelty[9] auch auf dem Prinzip, dass eine unerfüllte Klausel C_j aus der Klauselmenge $F = \{C_1, ..., C_n\}$ zufällig gewählt wird und anschließend eine Variable $v_{i,j}$ aus C_j geflippt wird. Dennoch unterscheiden sich beide Algorithmen davon, dass bei Novelty eine Variable $v_{i,j}$ aus C_j nach einem Wert t_i ausgewählt wird. Der Wert t_i beschreibt die Anzahl an erfüllten Klauseln, die durch einen Flip von Variable v_i ausgelöst wird. Das heißt aus der Klausel C_j wird die Variable $v_{i,j}$ mit dem größten Wert t_i gewählt.

Beispiel 7

Gegeben sei folgende Klauselmenge F:

$$\{\underbrace{\{v_1, \neg v_2, \neg v_3\}}_{C_1}, \underbrace{\{\neg v_1, v_2, \neg v_3\}}_{C_2}, \underbrace{\{v_1, v_2, \neg v_3\}}_{C_3}\}$$

Annahme: zufällige Startbelegung: $v_1 = 0, v_2 = 1, v_3 = 1$

- C_1 wird ausgewählt, da C_2 und C_3 erfüllt sind
- Für jede Variablen $v_{i,1}$ aus C_1 wird der Wert t_i berechnet
  1. Wenn $v_{1,1}$ geflippt wird $\rightarrow$ 3 Klauseln sind erfüllt
  2. Wenn $v_{2,1}$ geflippt wird $\rightarrow$ 2 Klauseln sind erfüllt
  3. Wenn $v_{3,1}$ geflippt wird $\rightarrow$ 3 Klauseln sind erfüllt
- Die Variablen der Klausel C_1 werden absteigend nach t_i sortiert:

$$\{v_1, \neg v_3, \neg v_2\}$$

Nachdem C_j sortiert wurde, wird mit einer Wahrscheinlichkeit $r \leq 1-p$ die Variable mit dem größtem Wert t_i gewählt. p ist eine feste Wahrscheinlichkeit, welches zu

Beginn zufällig aus $[0;1]$ gewählt wird. Dagegen wird r zu jedem Zug neu zufällig aus $[0;1]$ bestimmt. Falls $r \leq 1-p$ nicht eintritt, so wird die zweitbeste Variable aus C_j genommen.

5.1.4 CCLS

Im Gegensatz zu den SLS Algorithmen ist *Configuration Check Local Search* (kurz: CCLS) eine lokale Suche, welches anhand der Konfiguration einer Variable v_i entscheidet, ob v_i geflippt wird[3].

Definition 5.1.4.1 (Konfiguration) *Gegeben sei eine Klauselmenge $F = \{C_1, ..., C_n\}$. Außerdem sei die Menge $N(v_i)$, die alle Nachbarn von v_i enthält. Zwei Variablen v_i und v_j sind Nachbarn, wenn sie mindestens einmal zusammen in einer Klausel $C_k \in F$ $(1 \leq k \leq n)$ vorkommen.*
Die Konfiguration einer Variablen v_i besteht aus den Belegungen von $N(v_i)$ und wird als einen Vektor K_{v_i} bezeichnet.

Um zu entscheiden, welche Variable geflippt werden darf, wird folgende Funktion f definiert:

$$f(v_i) = \begin{cases} 1, falls & \overline{K_{v_i}} \neq K_{v_i} \\ 0, sonst & \end{cases}$$

Sei $\overline{K_{v_i}}$ die Konfiguration seit dem letzen Flip von v_i und K_{v_i} die Konfiguration im jetzigen Schritt. Eine 1 bedeutet, dass die Variable v_i geflippt werden darf. Im umgekehrten Fall ist eine 0, dass das Flippen der Variablen v_i verboten ist, da sich die Konfiguration K_{v_i} seit dem letztem Flip von v_i nicht geändert hat. Mit dieser Funktion f soll vermieden werden, dass eine Variable öfters geflippt wird. Falls v_i geflippt wird, so wird K_{v_i} auf 0 gesetzt. Falls jedoch K_{v_i} sich verändert, das heißt ein Nachbar von v_i wird geflippt, so wird K_{v_i} wieder auf 1 gesetzt[12].
Ein Nachteil von CCLS wie auch bei anderen lokalen Suchen ist es, dass die Lösung abhängig von der Startkonfiguration ist. Aus dem Grund werden vermutlich bessere Lösungen außerhalb des Suchraumes nicht gefunden.

5.2 UCTMAXSAT: Neuer Ansatz für MAXSAT

SLS Algorithmen sind nicht in der Lage zu entscheiden, ob eine Lösung optimal ist, welche aber dennoch in der Praxis oft genutzt werden, da sie schnelle Lösungen liefern[6]. Dieser neue Ansatz UCTMAXSAT soll optimale Lösungen liefern und dafür sorgen, dass der Algorithmus nicht stagniert. Hier in dem Fall ist Stagnation gemeint, dass die Wahl eines Knotens i uneindeutig ist, da die Knoten gleiche Werte haben. Somit wird keine vermutlich bessere Lösung gefunden, wenn der Algorithmus hängen bleibt. Dabei wird die MCTS Methode *Upper Confidence Bounds For Tree* (siehe Abschnitt 3.3.1) genutzt, weil dies als Banditen-Problem (siehe Abschnitt 2.2.2) angesehen werden kann. Der Exploitation-Teil ist das Finden einer optimalen Lösung und der Exploration-Teil ist das Vermeiden eines Verharrens des Algorithmus[6]. Außerdem ist UCTMAXSAT eine Kombination aus einer MCTS Methode und eines SLS Algorithmus, denn UCTMAXSAT nutzt den SLS Algorithmus für den Simulationsschritt.
In den nächsten Unterabschnitten wird die Funktionsweise des UCTMAXSAT Algorithmus aus [6] vorgestellt. Ebenfalls wird die Baumstruktur des UCTMAXSAT Algorithmus dargestellt.

5.2.1 Funktionsweise

Gegeben sei eine Klauselmenge $\varphi = \{C_1, ..., C_n\}$. Dazu wird auf ungewichtete Klauseln beschränkt. Ungewichtet bedeutet, dass jede Klausel gleich behandelt wird, unabhängig davon, welches Gewicht eine Klausel hat. Sei V die Menge der Variablen, die in φ vorkommen. Jeder Knoten i repräsentiert eine Variable v_i. Mit s_i wird ein Kindknoten des Knotens i beschrieben. In jedem Knoten i wird die Anzahl an Simulationen n_i und der geschätzte Durchschnittswert $\overline{x_i}$ gespeichert, wobei $n_i = 1$ und $\overline{x_i} = 0$ als Startwerte initialisiert werden. Außerdem wird eine Belegung ρ_{best} gespeichert, welches die optimale Belegung für φ ist. Sei T der Spielbaum (beziehungsweise wie in Abschnitt 3.2 bezeichnet als *internal tree*). In T wird also der Spielbaum nach und nach erweitert.
Der UCTMAXSAT Algorithmus wird in den vier einzelnen MCTS Schritten dargestellt, sodass die Anwendung des MCTS aus Kapitel 3 deutlicher wird.

1.) Selektion

Sei $I = \{1, ..., m\}$ die Menge aller Knoten im Baum T. Zu Beginn von UCTMAXSAT wird zufällig eine Belegung ρ_{best} gewählt und ein festes Limit k an Simulationen festgelegt. ρ_{best} ist zu jedem Zeitpunkt $t \leq k$ die aktuell optimale Belegung für φ.

Ebenfalls gibt es eine Menge ρ, die anfangs leer ist und die Belegungen der Variablen v_i von der Wurzel bis zu einem Blatt speichert. Das heißt, dass nur Variablen aus $\rho_{best} \cup \rho$ geflippt werden dürfen, welche nicht im Widerspruch zu ρ stehen. Ebenfalls werden in den Blättern b mit der Belegung aus ρ die Formel φ ausgewertet. Anschließend wird ein Knoten i erstellt und i wird in den leeren Spielbaum T hinzugefügt. Dieser Knoten i enthält eine Variable v_i, die mithilfe einer Heuristik ausgewählt wird. Hier in diesem Fall wird die Heuristik $A(0)$ genutzt, wobei $A(0)$ die Variablen v_i wählt, welche in den Klauseln am häufigsten vorkommen. Die 0 in $A(0)$ bedeutet, dass es keine Gewichte für die Klauseln gibt.[2]
Jeder Knoten $i \in T$ wird zu Beginn als *open* markiert, wobei *open* bedeutet, dass ein Knoten i sich noch in der Ausführung befindet, also nicht abgeschlossen ist[3]. Ist i abgeschlossen, so wird es als *closed* markiert. In den meisten Fällen werden Mengen genutzt, um *open*- und *closed*-Knoten zu speichern. Damit wird der Aufwand erspart für jeden Knoten i zusätzlich einzelne Werte zu speichern.
Bei der Selektion wird ein Kindknoten s_i eines Knotens i ausgewählt, welches den größten UCT-Wert $u(s_i)$ aufweist. Sei J die Menge der Kindknoten von i.

$$u(s_i) = argmax_{s_i \in J} \left(\overline{x_{s_i}} + C \cdot \sqrt{\frac{ln(n_i)}{n_{s_i}}} \right)$$

2.) Expansion

Jeder Knoten $i \in T$ besitzt stetig zwei Kinder (mit Ausnahme der Blätter), die als i_l (linkes Kind) und i_r (rechtes Kind) beschrieben. Ebenfalls gibt es ausgehend von einem Knoten i immer zwei Kanten, welches die Variable v_i auf $falsch$ ($\rho \leftarrow \rho \cup \{\neg v_i\}$) oder $wahr$ ($\rho \leftarrow \rho \cup \{v_i\}$) setzt. Wählt man $falsch$, so gelangt man zum linken Kind i_l und ansonsten zum rechten Kind i_r.
Nachdem ein Knoten i mithilfe von UCT ausgewählt wurde, werden die Kindknoten i_l und i_r expandiert, das heißt die Kindknoten werden in T hinzugefügt. Daraus folgt, dass bei jedem Selektionsschritt immer 2 Kindknoten dem aktuell ausgewählten Knoten i im Baum T angehängt werden.

3.) Simulation

In diesem Schritt kommt der SLS Algorithmus zum Einsatz. Denn normalerweise wird bei MCTS nur ein Playout durchgeführt. Bei UCTMAXSAT werden zwei Playouts durchgeführt. Diese Playouts werden mithilfe von einem ausgewählten SLS

[2] Zu $A(0)$: Klauselmenge $\{\{v_1, v_2\}, \{v_1, \neg v_3\}\} \rightarrow A(0)$ wählt v_1
[3] Oder: Knoten i wurde noch nicht besucht

Algorithmus W ausgeführt. Wichtig hierbei ist, dass jede Variable v_i nur einmal geflippt werden darf. Das heißt, wenn v_i in Schritt t geflippt wurde, dann darf v_i nie wieder geflippt werden. Somit wird das Zyklenproblem verhindert.
Eine vollständige Belegung für ρ liegt in den Blättern vor. In einem Blatt b wird die Anzahl an erfüllten Klauseln für φ mithilfe von ρ berechnet. ρ beinhaltet alle Belegungen von der Wurzel bis zu einem Blatt b[4]. Die Anzahl an erfüllten Klauseln wird mit f ausgedrückt. Dabei ist f_r der erreichte Wert der Simulation des rechten Kindes und f_l für das linke Kind. Falls das linke Kind oder das rechte Kind kein Blatt ist, so wird mit ρ_{best} der Wert für f_l beziehungsweise f_r ermittelt. Die Knoten, die in der *Search Policy* sind und beim Playout durchlaufen werden, werden dem Spielbaum (internal tree) nicht hinzugefügt. Es kann durchaus geschehen, dass die Differenz von f_r und f_l sehr groß sein kann. Aus dem Grund wird der Mittelwert für f ermittelt:

$$f = \frac{f_l^2 + f_r^2}{2}$$

Der Grund für das Quadrieren ist, dass die Suche auf wenige *Plateaus* stoßen soll. Plateaus sind Knoten, welche denselben Wert besitzen[15]. Dies würde zur Uneindeutigkeit bei der Wahl eines Knotens führen.

4.) Backpropagation

Nachdem die beiden Simulationen durchgeführt wurden, werden die Werte der Knoten s' im Spielbaum, die besucht wurden (Knoten aus der Search Policy sind ausgeschlossen), neu berechnet. Für alle Knoten s' wird die Anzahl an Simulationen $n_{s'}$ um 1 erhöht. Außerdem wird der Durchschnittswert $\overline{x_{s'}}$ mit folgender Berechnung neu berechnet:

$$\overline{x_{s'}} \leftarrow \overline{x_{s'}} + \frac{f - \overline{x_{s'}}}{n_{s'}}$$

Zum Schluss werden die besuchten Blätter als *closed* markiert, da in den nächsten Simulationen neue vermutlich bessere Lösungen erkundet werden sollen.

[4]Sei die Wurzel r und ein Blatt b. Dann enthält ρ alle Belegungen der Variablen auf dem Weg $(r, ..., b)$

Pseudocode

Hier im Folgenden ist der Pseudocode des UCTMAXSAT Algorithmus aus Goffinet[6]. Unter den Listings befinden sich die Kommentare zu dem Code aus [6].

```
1  Algorithm 1 UCTMAXSAT main loop
2  ─────────────────────────────────────────────────────────────
3  globals:
4   φ: the input (non-empty) CNF Formula
5   ρ_best ← a random complete assignment        ▷ (1)
6
   function UCTMAXSAT
8   root ← CREATE_NODE(∅)      ▷ (2)
9   repeat
10    DESCEND_NODE(root,φ,∅)
11  until time is met
12  return ρ_best
13 end function
```

(1) = best solution found so far
(2) = pointer to root node of search tree

```
1  Algorithm 2 UCTMAXSAT helper routines
2  ─────────────────────────────────────────────────────────────
3  function CREATE_NODE(ρ)
4   node ← new node
5   node.visits ← 1
6   ρ' ← set of literals in ρ_best that do not contradict a
       literal in ρ
7   Run SLS Algorithm using ρ ∪ ρ_best as starting configuration,
       never flipping variables in ρ
8   f ← fraction of satisfied clauses in best assignment
       found by SLS run
9   node.value ← f^2
10  Update ρ_best if the SLS run uncovered a new best solution
11  return node
12 end function
13
   function DESCEND_NODE(node,φ',ρ)          ▷ (3)
```

```
15  if ρ is a complete assignment then
16   Mark node as closed
17   f ← fraction of clauses in φ that are satisfied by ρ
18   return f^2
19  else if node.visits = 1 then   ▷ (4)
20   node.variable ← A(0)
21   node.left ← CREATE_NODE(ρ ∪ {¬node.variable})
22   node.right ← CREATE_NODE(ρ ∪ {node.variable})
23   r ← (node.left.value + node.right.value)/2
24  else
25   child, φ', ρ ← SELECT_CHILD(node, φ', ρ)
26   r ← DESCEND_NODE(child, φ', ρ)
27  end if
28   node.visits ← node.visits + 1
29   node.value ← node.value + (r − node.value)/node.visits
30   Mark node as closed if node.left and node.right are both
        closed
31  return v
32 end function
33
   function SELECT_CHILD(node, φ', ρ)
35  if either node.left or node.right is closed then
36   ch ← the still open child
37  else
38   ch ← child that maximizes UCB1 score
39  end if
40  if ch is the left child of node then
41   φ' ← φ' simplified by assigning node.variable to FALSE
42   ρ ← ρ ∪ {¬node.variable}
43  else
44   φ' ← φ' simplified by assigning node.variable to TRUE
45   ρ ← ρ ∪ {node.variable}
46  end if
47  return (ch, φ', ρ)
48 end function
```

(3) = ρ is the current partial assignment (set of literals)

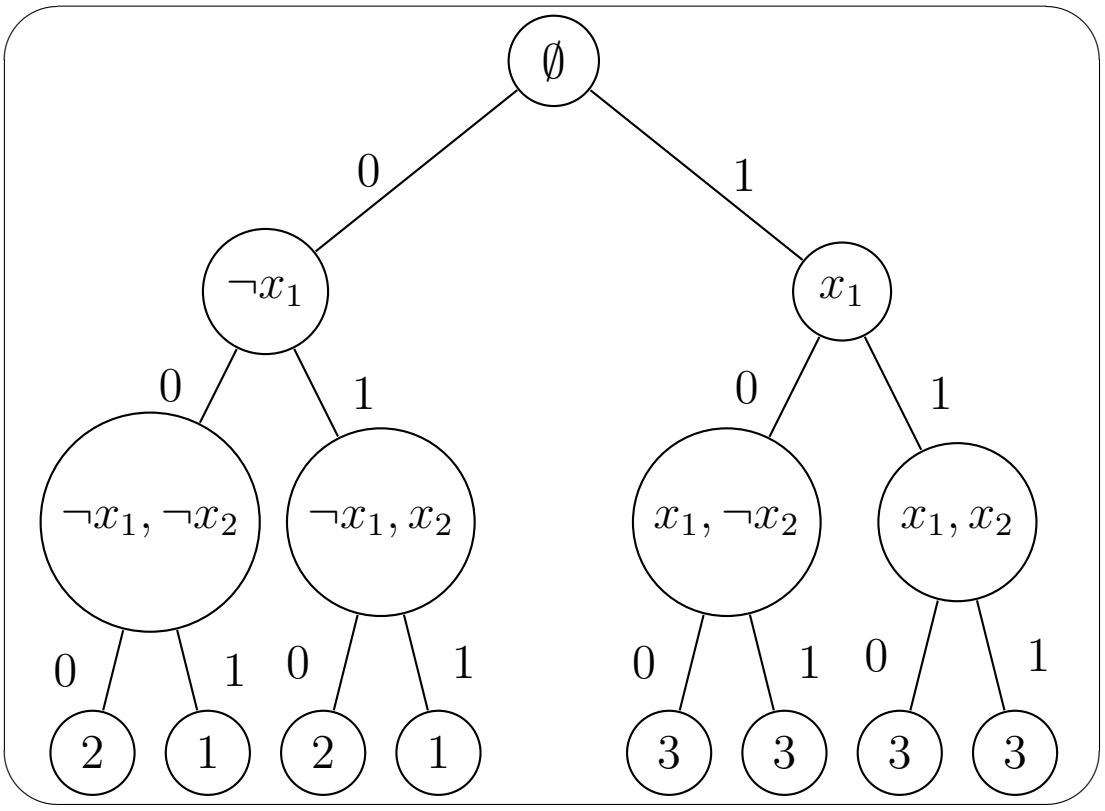
(4) = a previously unexpanded node

5.2.2 Baumstruktur

In diesem Abschnitt wird die Baumstruktur betrachtet. Dabei wird nicht die Perspektive der MCTS Schritte dargestellt, sondern aus der Sicht wie der vollständige Baum für eine gegebene CNF φ aussehen würde.
Gegeben sei folgende CNF φ:

$$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_2)$$

Für φ sieht der vollständige Spielbaum folgendermaßen aus:



An der Wurzel wird mit $\rho = \{\}$ gestartet. Mithilfe der Heuristik $A(0)$ wird zunächst die Variable x_i gewählt, die am häufigsten in allen Klauseln vorkommt. Das heißt, in diesem Fall ist es die erste Variable x_1. Zu jedem Knoten existieren zwei Kanten. Eine 0 bedeutet, dass die ausgewählte Variable x_i auf $falsch$ gesetzt wird, das heißt $\rho = \{\neg x_i\}$. Dagegen ist eine 1, dass x_i auf $wahr$ gesetzt wird, also $\rho = \{x_i\}$. Dies wird bis zu einem Blatt b ausgeführt. Ein Blatt b enthält eine vollständige Belegung von ρ und gibt als einen Wert f die Anzahl an erfüllten Klauseln zurück. In diesem Fall würde beim Backpropagation-Schritt der Wert f quadriert werden wie in Abschnitt 5.2.1 bereits beschrieben wurde.

5.3 Untersuchung von UCTMAXSAT

In diesem Abschnitt wird UCTMAXSAT unter verschiedenen Aspekten untersucht. Dabei steht der Monte-Carlo Tree Search als Hauptschwerpunkt im Vordergrund.

5.3.1 Optimale Lösung im Suchbaum

Ziel von UCTMAXSAT ist es, für eine gegebene Klauselmenge φ eine Belegung ρ zu finden, sodass MAXSAT gelöst wird. MAXSAT ist genau dann gelöst, wenn mithilfe von ρ eine größtmögliche Anzahl π an Klauseln $C_j \in \varphi$ erfüllt sind. Im best case würde $\pi = |\varphi|$ gelten, das heißt alle Klauseln $C_j \in \varphi$ sind erfüllt. Somit wäre das MAXSAT und das SAT Problem gelöst. Im worst case ist $\pi = 0$. Es existiert eine Belegung ρ, sodass alle Klauseln C_j unerfüllt sind.

Beispiel 8

Gegeben sei die folgende CNF Formel φ:

$$\varphi := (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$$

$\Rightarrow$ Es existiert keine Belegung ρ, sodass alle Klauseln unerfüllt sind. Es ist immer mindestens eine Klausel C_j erfüllt.
Für die folgende CNF φ_2 gibt es eine Belegung ρ, sodass $\pi = 0$:

$$\varphi_2 := (x_1 \vee x_2) \wedge (x_2 \vee x_3)$$

$\Rightarrow$ Wenn $x_1 = x_2 = x_3 = 0$ ist, dann beträgt $\pi = 0$.

Durch die Nutzung des MCTS in UCTMAXSAT wird eine optimale Lösung approximativ gefunden. MCTS ist eine Bestensuche und wählt den als nächsten zu expandierenden Knoten i mithilfe einer Heuristik h. Bei UCTMAXSAT wird die Heuristik *Upper Confidence Bounds For Trees* (siehe Abschnitt 3.3.1) verwendet, die den jeweiligen Knoten v_i wählt, der den größten Wert u_i wählt. Dadurch wird verhindert, dass der Algorithmus in einem lokalen Maximum hängen bleibt, da es anscheinend eine bessere Lösung gibt. Außerdem wird mittels UCT das Exploration-Exploitation Dilemma mitberücksichtigt. Denn durch eine geschickte Wahl des Koeffizienten C wird die Balance zwischen vielversprechenden und weniger vielversprechenden Knoten i geregelt. Anhand der *open*- und *closed*-Mengen werden bereits besuchte Teilbäume nicht erneut besucht. Das heißt, dass UCT nur den Knoten i

wählt, der in der *open*-Menge enthalten ist. So werden auch unentdeckte eventuell bessere Lösungen besucht.
Sei g das Limit an Ausführungen der MCTS Schritte. Das heißt, falls die MCTS Schritte g-mal ausgeführt wurden, so wird der Algorithmus beendet und es wird die aktuell beste Lösung ρ_{best} zurückgegeben. Durch die Wahl von g kann die Qualität des Algorithmus beeinflusst werden. Denn je höher g ist, desto genauer werden die Durchschnittswerte $\bar{x}_i$. Demnach ergibt sich durch eine höhere Anzahl an Simulationen, dass die Näherung eines optimalen Zuges verbessert wird. Insgesamt gibt es $2 \cdot g$ Simulationen, da in jedem Durchgang zwei Simulationen durchgeführt werden.

5.3.2 Laufzeitanalyse und Speicheraufwand

Bei einem Durchlauf $d \leq g$ von UCTMAXSAT werden alle 4 Schritte (siehe Abschnitt 5.2.1) ausgeführt. Mit c wird die Verzweigungsrate, also die Anzahl an Kindknoten bezeichnet. Daraus folgt, dass jeder Knoten i eine Verzweigungsrate c von 2 hat. Zu Beginn von UCTMAXSAT wird ein Knoten i, der die Wurzel bildet, in den Baum T hinzugefügt. Auf diesem Knoten wird ein SLS Algorithmus W ausgeführt. Im ersten Durchlauf d werden zwei Kindknoten i_l und i_r in T hinzugefügt. Daraufhin wird ebenfalls auf diesen beiden Knoten zwei parallele Simulationen mithilfe von W ausgeführt. Das heißt für jeden Durchlauf $d \leq g$ werden immer zwei Knoten expandiert und zwei Simulationen parallel durchgeführt. Es kann davon ausgegangen werden, dass für jede Simulation ein Thread *thr* genutzt wird. Somit gibt es zwei Threads bei UCTMAXSAT. Daraus ergibt sich folgende Laufzeit für UCTMAXSAT:

$$
\begin{aligned}
&\Rightarrow \overbrace{1 + O(W)}^{Wurzel+SLS} + g \cdot (c + \frac{2 \cdot O(W)}{thr}) \\
&= 1 + O(W) + g \cdot (c + \frac{2 \cdot O(W)}{2}) \\
&= 1 + O(W) + g \cdot (c + O(W)) \\
&= 1 + O(W) + g \cdot c + g \cdot O(W) \\
&= g \cdot O(W) \cdot (c + 1) + 1 \\
&= g \cdot O(W) \cdot (2 + 1) + 1 \\
&= g \cdot O(W) \cdot 3 + 1 \\
&\Rightarrow O(g \cdot O(W))
\end{aligned}
$$

Es ist zu erkennen, dass die Laufzeit von dem Limit an Durchläufen g und vom SLS Algorithmus abhängig ist. Dennoch muss g geschickt gewählt werden, denn

wenn g größer ist, ist zwar die Laufzeit langsamer, aber dafür ist die Näherung eines optimalen Zuges besser. Die Laufzeit von W kann sehr verschieden ausfallen. Beispielsweise arbeitet CCLS in linear Zeit, woraufhin WalkSAT eine quadratische Laufzeit mit dem Algorithmus aus 5.1.2 haben kann.
Wie schon bereits erwähnt, werden bei jedem Durchlauf d immer zwei Knoten hinzugefügt. Damit würde der Speicheraufwand $O(c \cdot g)$ betragen. Hingegen ist im worst case der Baum T in Form eines vollständigen Binärbaums. Damit wäre der Speicheraufwand exponentiell, denn es gibt in jeder Ebene des Baumes 2^t viele Knoten. Also beläuft sich der Speicheraufwand auf $O(2^t)$. Zusammenfassend ergibt sich folgende Tabelle:

Eigenschaft	Worst Case	Best Case
Laufzeit	$O(g \cdot O(W))$	$O(g \cdot O(W))$
Speicherplatz	$O(2^t)$	$O(g \cdot c)$

5.3.3 Vergleich UCTMAXSAT und bisherige Solver

Dieser Unterabschnitt stellt den Vergleich zwischen einem SLS Algorithmus W und die UCTMAXSAT(W)-basierte Variante dar. UCTMAXSAT(W)-basierte Variante bedeutet, dass UCTMAXSAT für die Durchführung der Simulationen einen SLS Algorithmus W verwendet.
Obwohl in ρ die bisher expandierten Variablen v_i festgehalten werden und nicht wieder geflippt werden dürfen, kann es dennoch dazu kommen, dass Zyklen entstehen. Denn der SLS Algorithmus W verwendet als Startkonfiguration $\rho_{best} \cup \rho$ (Variablen, die im Widerspruch mit ρ sind, sind nicht enthalten) und das Flippen ist abhängig vom jeweiligen SLS Algorithmus W. Damit ist gemeint, dass das Zyklenproblem in W selbst auftritt, aber keinen Einfluss auf ρ hat. Es sollte im Klaren sein, dass bei UCTMAXSAT(W) und W das Zyklenproblem weiterhin besteht.
Ein großer Unterschied zwischen UCTMAXSAT und den SLS Algorithmen ist es, dass in UCTMAXSAT mehr Wissen über eine optimale Lösung zur Verfügung steht. Denn in jedem Knoten i werden Durchschnittswerte gespeichert, welche über die erwartete Anzahl an erfüllten Klauseln Auskunft geben. Hingegen gibt es bei den SLS Algorithmen nur eine Vergleichsfunktion, die nur für den Fall überprüft, ob eine gegebene CNF φ erfüllt ist. Es kann auch der Fall sein, dass φ nicht erfüllt ist und die maximale Anzahl an erfüllten Klauseln ermittelt werden soll.

UCTMAXSAT(W) vs. WalkSAT

WalkSAT arbeitet mit dem einfachen Prinzip, dass mit einer bestimmten Anzahl an Versuchen zufällig Variablen aus unerfüllten Klauseln geflippt werden. Es existiert

kaum eine Strategie, um eine optimale Lösung zu finden. Dagegen nutzt die Kombination mit UCTMAXSAT eine genauere Strategie anstatt die Suche dem Zufall zu überlassen. Wie schon oben erwähnt, besitzt WalkSAT nur eine einzige Vergleichsfunktion, aber diese Funktion dient nur für den Fall, dass alle Klauseln erfüllt sind. Der Speicherbedarf von WalkSAT ist linear, denn es wird in einer Liste ein Paar (ρ, Anzahl an erfüllten Klauseln) gespeichert. Der Speicherbedarf von UCTMAXSAT ist höher, da ein Baum T erstellt wird, indem jeder Knoten Informationen wie den Durchschnittswert und die Anzahl an Simulationen enthält. Für die Laufzeit gibt es 2 Fälle:

1. Mit maximaler Anzahl an Flips $m \rightarrow$ WalkSAT läuft dann in[5] $O(k \cdot m)$ und somit hätte UCTMAXSAT(WalkSAT) eine Laufzeit von
   $O(g \cdot O(W)) = O(g \cdot k \cdot m)$

2. Ohne maximale Anzahl an Flips $m \rightarrow$ WalkSAT würde in linearer Zeit laufen und dagegen hätte UCTMAXSAT(WalkSAT) eine Laufzeit von $O(g \cdot O(W)) = O(g \cdot k)$

Somit ist WalkSAT schneller. Obwohl UCTMAXSAT(WalkSAT) nicht schneller ist und mehr Platz verbraucht, liefert es in den meisten Fällen bessere Ergebnisse als WalkSAT. Das heißt, die Anzahl an erfüllten Klauseln ist bei UCTMAXSAT(WalkSAT) höher als WalkSAT. Dies erkennt man an den Resultaten (von Goffinet und Ramanujan) in der folgenden Tabelle[6]:

	Random Instances			Crafted Instances		
	$\sigma > 0$	$\sigma = 0$	$\sigma < 0$	$\sigma > 0$	$\sigma = 0$	$\sigma < 0$
WalkSAT	393 (56.5%)	287 (41.3%)	15 (2.2%)	336 (83.6%)	63 (15.7%)	3 (0.7%)
Novelty	364 (52.4%)	309 (44.5%)	22 (3.1%)	311 (77.4%)	76 (18.9%)	15 (3.7%)

Die Ergebnisse wurden mithilfe der *normalisierten Differenz* berechnet, wobei s_1 das Ergebnis von UCTMAXSAT(WalkSAT) darstellt und s_2 hingegen das Ergebnis für WalkSAT ist[6]:

$$\sigma(s_1, s_2) = \frac{s_1 - s_2}{max\{s_1, s_2\}}$$

In der Tabelle bedeutet $\sigma > 0$, dass UCTMAXSAT(WalkSAT) bessere Ergebnisse liefert. Man kann anhand der Tabelle folgern, dass die Suche durch das Einbringen

[5] Wiederholung: k ist die begrenzte Anzahl an Versuchen für WalkSAT

von mehr Wissen verbessert wird und somit eine bessere Lösung gefunden wird. Damit sind die Lösungen von UCTMAXSAT(WalkSAT) vielversprechender als die von WalkSAT.

UCTMAXSAT(Novelty) vs. Novelty

Novelty und WalkSAT unterscheiden sich nur in dem Punkt, dass bei Novelty die Variablen der ausgewählten unerfüllten Klausel C_j sortiert werden und anschließend mit einer Wahrscheinlichkeit $r \leq 1 - p$ die beste Variable geflippt wird. Das heißt, die Performanz von Novelty (wie man auch in der Tabelle sieht) ist höher als die von WalkSAT. Novelty benötigt mehr Speicher, denn für die Sortierung werden zusätzlich Werte t_i (siehe 5.1.3) gespeichert, wodurch Novelty an Laufzeit verliert.

UCTMAXSAT(CCLS) vs. CCLS

Der Speicherbedarf von CCLS ist gegenüber UCTMAXSAT(CCLS) etwas geringer, denn es wird nur ein Array A benötigt, welches für jede Variable v_i einen Wert speichert, wobei 1 bedeutet, dass die Konfiguration N_{v_i} von v_i sich geändert hat und bei einer 0 nicht. Dennoch ist das Speichern der Konfigurationen N_{v_i} aufwendig und redundant, denn es müssen alle Variablen einer Klausel C_j gespeichert werden, die mit v_i benachbart sind. Beispielsweise wenn v_1 in jeder Klausel C_j vorhanden ist, muss in jeder Konfiguration v_1 gespeichert werden.
Von allen drei vorgestellten SLS Algorithmen liefert CCLS die besten Ergebnisse. Außerdem auf zufälligen 3-SAT Formeln schneidet UCTMAXSAT schlechter ab als CCLS. Jedoch ergeben sich für UCTMAXSAT bei vorher-festgelegten SAT Formeln bessere Resultate.

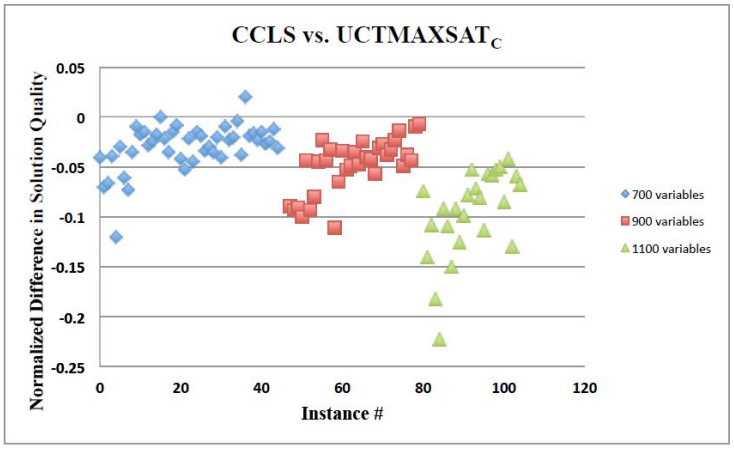


Abbildung 5.1: Zufällige 3-SAT Formeln[6]

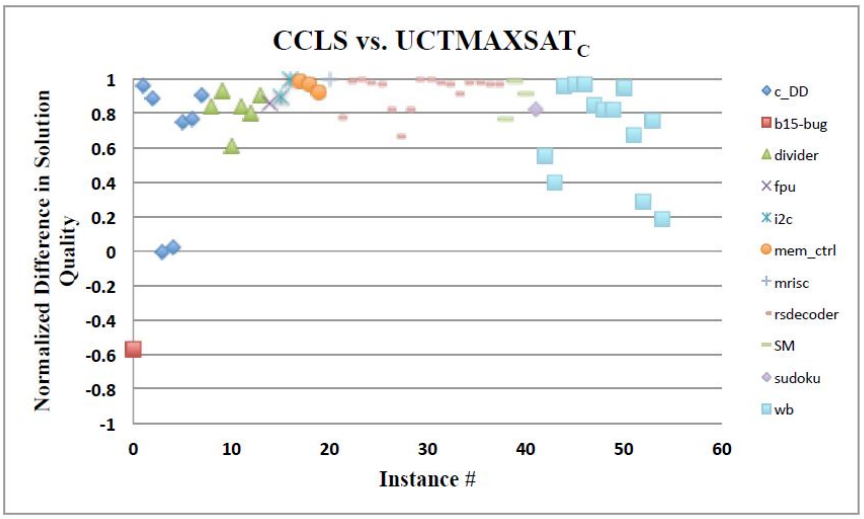


Abbildung 5.2: Vorher-festgelegte SAT Formeln[6]

5.3.4 Vor- und Nachteile von UCTMAXSAT

Im vorherigen Abschnitt wurde UCTMAXSAT mit den SLS Algorithmen aus 5.1 verglichen. Im Folgenden werden wesentliche Vor- und Nachteile von UCTMAXSAT aufgelistet. Es ergeben sich folgende Vorteile für UCTMAXSAT:

- Mithilfe des Suchbaums können Lösungen effizienter auf Optimalität überprüft werden. Dank der Heuristik UCT wird mehr Wissen zur Verfügung gestellt, weshalb die suboptimalen Knoten nicht mehr betrachtet werden. Damit wird das Banditen-Problem einigermaßen gelöst. Aufgrund des Koeffizienten C gibt es eine Balance zwischen optimalen und weniger optimalen Knoten.

- Es wird keine statische Bewertungsfunktion benötigt. Wie man bei UCTMAXSAT sieht, musste für die UCT Heuristik nur die Variablenbezeichnungen verändert werden (je nachdem wie die Knoten festgelegt wurden). Demnach kann man MCTS in vielen Spielen (oder wie hier: bei theoretischen Problemen) einsetzen.

- Aus dem ersten Punkt ist ebenfalls zu folgern, dass nicht alle Knoten beziehungsweise Teilbäume besucht werden. Bei UCTMAXSAT ist dies jedoch abhängig von UCT und der Anzahl an Durchläufen g von MCTS-Schritten.

- Verwendung von SLS Algorithmen für die Ausführung eines Playouts, da sie schnell und platzsparend sind. Außerdem muss nur der Wert des SLS Algorithmus zurückgegeben werden, um Backpropagation durchzuführen. Das heißt, es muss für einen zufälligen Playout nicht die ganze Tiefe durchlaufen werden.

- Parallele Ausführung von 2 SLS Playouts.
  Normalerweise wird nur ein Playout (siehe Kapitel 3) ausgeführt, aber hier werden stets zwei Playouts ausgeführt. Damit wird die Laufzeit verbessert und somit wächst auch der Baum schneller, sodass mehr Informationen vorhanden sind.

- Vermeiden von Zyklen bisher expandierter Knoten.
  Es werden nur die Knoten bei einem Playout geflippt, welche nicht in ρ vorhanden sind. Bisher expandierte Knoten haben somit ihren „festen Platz“ im Suchbaum.

Hingegen gibt es für UCTMAXSAT folgende Nachteile:

- UCTMAXSAT arbeitet langsamer als die vorgestellten SLS Algorithmen und benötigt sehr viel Speicher. Dies liegt vor allem an der Konstruktion des Spielbaums und die darin enthaltenen Informationen wie zum Beispiel der Durchschnittswert $\bar{x}_i$, die Anzahl der Simulationen n_i und ρ.

- Der SLS Algorithmus W für die Playouts beeinflusst die Laufzeit von UCTMAXSAT und die Optimalität einer Lösung. Wenn der Zufall beim SLS Algorithmus „die richtige Wahl“ trifft [6], so ergibt sich ein besserer Wert f. Deshalb ist dieser Punkt umstritten, ob SLS Algorithmen für die Simulation genutzt werden sollen.

- Unvollständig, da nicht immer eine Lösung gefunden wird

[6] In anderen Worten: Der Zufall trifft eine „richtige Wahl“, wenn durch einen Flip einer Variable eine höhere Anzahl an erfüllten Klauseln erreicht wird.

Kapitel 6

Fazit

In dieser Arbeit wurde der neue Ansatz UCTMAXSAT untersucht, der aus einer Kombination zwischen Monte-Carlo Tree Search Methode und MAXSAT besteht. Um sich mit der Thematik dieser Arbeit vertraut zu machen, wurden zu Beginn Grundlagen wie Spieltheorie, Entscheidungstheorie und Monte-Carlo Methoden vorgestellt. Anschließend wurde der Monte-Carlo Tree Search behandelt. Das Grundprinzip wurde anhand der vier Schritte: Selektion, Expansion, Simulation und Backpropagation erklärt. Ebenfalls wurde es einen Überblick zu den Heuristiken gegeben, da diese für den MCTS eine große Rolle spielen. Im Anschluss wurde auf das MAXSAT-Problem eingegangen.
Nachdem der Grundbaustein dieser Arbeit gelegt wurde, wurde als Nächstes der Hauptschwerpunkt thematisiert. Der Hauptschwerpunkt dieser Arbeit ist die Untersuchung von MCTS mit Anwendung auf MAXSAT. Zunächst wurden existierende Solver und deren Lösungen des MAXSAT-Problems vorgestellt. Um es explizit auszudrücken, bestehen die Solver aus SLS Algorithmen. Dabei wurde zur Kenntnis genommen, dass SLS Algorithmen in Hinsicht auf MAXSAT Probleme aufweisen. Beispielsweise ist in jedem der hier vorgestellten SLS Algorithmen das Zyklenproblem vorhanden. Außerdem wird hauptsächlich nur eine lokale Lösung gefunden, die nicht in allen Fällen optimal ist.
Um Abhilfe zu schaffen, wurde der neue Ansatz UCTMAXSAT aus Goffinet und Ramanujan vorgestellt. Dabei wurde im Detail auf die Funktionsweise und die Baumstruktur eingegangen. Daraufhin wurde UCTMAXSAT auf bestimmte Aspekte wie zum Beispiel Optimalität und Laufzeit analysiert. Wie sich herausstellte, benötigt UCTMAXSAT viel Speicher und besitzt eine langsame Laufzeit. Das liegt vor allem an der Nutzung eines besser informierten Suchbaums und es werden jegliche Informationen in den Knoten gespeichert. Damit wird der Suche viel mehr Wissen zur Verfügung gestellt. Dies spiegelt sich auch in den Resultaten (aus Goffinet und

Ramanujan) wieder. Damit ist gemeint, dass UCTMAXSAT gute Ergebnisse liefert und sich somit als einen guten Algorithmus darstellt. Wie zu sehen ist, kann Monte-Carlo Tree Search in vielen Bereichen verwendet werden. Bisher war MCTS nur im Spiel Go bekannt, aber jedoch wurde in dieser Arbeit bestätigt, dass MCTS nicht nur in Spielen zum Einsatz kommen kann, sondern auch in anderen Problemen, wie hier bei MAXSAT, verwendet werden kann. Ein Grund dafür sind die ausgezeichneten dynamischen Heuristiken. Vor allem die Heuristik UCT, welche in UCTMAXSAT verwendet wurde, löst größtenteils das Banditenproblem. Es werden dadurch nicht-optimale Lösungen ausgefiltert, das heißt, diese Lösungen werden bei der Suche nicht mehr betrachtet.
Zusammenfassend lässt sich ausdrücken, dass UCTMAXSAT in den meisten Fällen das MAXSAT-Problem besser als die vorgestellten SLS Algorithmen löst. Dennoch muss bemängelt werden, dass UCTMAXSAT langsam arbeitet und viel Speicher benötigt. Außerdem ist dieser Algorithmus unvollständig, weil nicht in allen Fällen eine optimale Lösung gefunden wird.

In Zukunft könnte man eventuell eine Variante entwickeln, die keinen SLS Algorithmus für die Simulation verwendet. Diese Variante könnte man dann mit UCTMAXSAT vergleichen. SLS Algorithmen sind recht schnell und benötigen wenig Speicher, aber sie sind nicht immer optimal. Ebenfalls könnte man UCTMAXSAT modifizieren, in dem an den Blättern eine Testfunktion ausgeführt wird. Diese Testfunktion überprüft, indem der Wert f eines Blattes mit der totalen Anzahl an erfüllten Klauseln übereinstimmt und den Algorithmus an dem Punkt beendet. Hier würde im best cace eine optimalere Laufzeit und eine effizientere Speicherung erreicht werden. Im Gegensatz dazu ergibt sich im worst case exponentiell viele Blätter und somit werden exponentiell viele Ausführungen der Testfunktion vollzogen.
Abschließend kann man sagen, dass Monte-Carlo Tree Search in vielen Problemen oder auch NP-vollständigen Problem eingesetzt werden kann, um diese Probleme approximativ zu lösen.

Literaturverzeichnis

[1] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. *Finite-time Analysis of the Multiarmed Bandit Problem.* 2002.

[2] Yossi Azar and Alon Ardenboim. *Algorithmic Methods Lecture: Approximations for MAX-SAT.* WiSe 2009/2010.

[3] Shaowei Cai and Kaile Su. *Configuration Checking with Aspiration in Local Search for SAT.* Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, 2012.

[4] Guillaume Maurice Jean-Bernard Chaslot. *Monte-Carlo Tree Search.* 2010.

[5] Amin Coja-Oghlan and Alan Frieze. *AnalyzingWalksat on Random Formulas.* Proceedings of ANALCO 2012, 2014.

[6] Jack Goffinet and Raghuram Ramanujan. *Monte-Carlo Tree Search for the Maximum Satisfiability Problem.* Springer International Publishing, Cham, 2016.

[7] GoGameGuru. *DeepMind AlphaGo vs Lee Sedol.* `https://gogameguru.com/tag/deepmind-alphago-lee-sedol/`, 2016 (Besucht am: 14.03.2017).

[8] Daniel Hennes and Dario Izzo, editors. *Interplanetary Trajectory Planning with Monte Carlo Tree Search.* International Joint Conference on Artificial Intelligence (IJCAI-15), 2015.

[9] Holger H. Hoos. *On the Run-time Behaviour of Stochastic Local Search Algorithms for SAT.* Proc. of AAAI-99, MIT Press, 1999.

[10] Wei Pan Kaifu Zhang. *The Two Facets of the Exploration-Exploitation Dilemma.* 2006.

[11] Prof. Dr. Kosfeld and Dr. Florian Hett. *Basiskurs Management (BWL-BMGT).* 2017.

[12] Chuan Luo, Shaowei Cai, Zhong Jie, and Kaile Su. *CCLS: Solver Description.* http://www.maxsat.udl.cat/13/solvers/ccls__.pdf (Besucht am: 16.04.2017).

[13] Joseph Antonius Maria Nijssen. *Monte-Carlo Tree Search for Multi-Player Games.* 2013.

[14] Stuart Russell, Siddharth Srivastava, and Aijun Bai. *Markovian State and Action Abstractions for MDPs via Hierarchical MCTS.* International Joint Conference on Artificial Intelligence (IJCAI-16), 2016.

[15] Prof. Dr. Manfred Schmidt-Schauss. *Einführung in die Künstliche Intelligenz Skript.* SoSe 16.

[16] Prof Dr. Manfred Schmidt-Schauss. *Logikbasierte Systeme der Wissensverarbeitung.* SoSe 16.

[17] Schnitger. *Internet Algorithmen Skript Kapitel 10 Algorithmische Spieltheorie.* http://www.thi.informatik.uni-frankfurt.de/lehre/ial/sose14/ial_sose14_skript.pdf, SoSe 14.

[18] Prof. Dr. Georg Schnitger. *Theoretische Informatik 1.* WiSe 2012/2013.

[19] Whos who the people lexicon. *Garri Kasparow.* http://www.whoswho.de/bio/garik-kimovich-weinstein.html, (Besucht am: 14.03.2017).

[20] Yingce Xia, Tao Qin, Weidong Ma, Nenghai Yu, and Tie-Yan Liu. *Budgeted Multi-Armed Bandits with Multiple Play.* International Joint Conference on Artificial Intelligence (IJCAI-16), 2016.

[21] Timothy Yee, Viliam Lis, and Michael Bowling, editors. *Monte Carlo Tree Search in Continuous Action Spaces with Execution Uncertainty.* International Joint Conference on Artificial Intelligence (IJCAI-16), 2016.